1. 上课须知
2. 上次课程内容
3. 上次作业
4. 本次课程内容预告
5. ZooKeeper 源码剖析
    5. 1. ZooKeeper版本选择
    5. 2. ZooKeeper 源码环境准备
    5. 3. ZooKeeper 序列化机制
    5. 4. ZooKeeper 持久化机制
    5. 5. ZooKeeper 网络通信机制
    5. 6. Zookeeper的Watcher工作机制
    5. 7. HDFS源码
    5. 8. ZooKeeper的集群启动脚本分析
    5. 9. ZooKeeper的QuorumPeerMain启动
    5. 10. ZooKeeper的冷启动数据恢复
    5. 11. ZooKeeper选举FastLeaderElection源码剖析
6. 总结

1. 上课须知

课程主题：ZooKeeper 第三次课 -- 源码剖析
上课时间：20:00 - 23:00
课件休息：21:30 左右 休息10分钟
课前签到：如果能听见音乐，能看到画面，请在直播间扣 666 签到

2. 上次课程内容

第一次课程的主要内容：

```
1、课程安排 + 上课约定
2、集中式 分布式 摩尔定律 去IOE 性能指标 一致性级别
3、分布式事务（2PC,3PC）
4、分布式一致性算法（Paxos，Raft，ZAB）
5、鸽巢原理 + NWR读写模型 + Quorum议会制
6、CAP 和 BASE 理论
7、ZooKeeper 介绍 和 设计目的
```

第二次课程的主要内容：

```
1、ZooKeeper核心功能
    znode数据模型
```

```
    watcher监听机制
    session会话机制
    zookeeper编程模型
2、zookeeper的应用场景
    发布订阅
    命名服务
    集群管理
    分布式锁
    队列管理
    分布式锁
3、企业实战案例
    分布式锁
    选举
    配置管理
    ......
```

3. 上次作业

作业：实现一个文件系统（类似于 zk 的数据模型）

验收要求：

```
1、实现基本的树形结构（DataTree），并且具备节点的增删改查等相关功能
    如果可以，帮我实现文件夹的剪切/移动
    DataTree dt = new DataTree()
    dt.insert(new DataNode())
2、具备冷启动数据恢复的功能
    具备把 内存中的数据  和 磁盘中的数据做同步和交换的 功能
    class ZKDatabase{
        private DataTree dt;
        private FileTxnSnapLog fileTxnSnapLog;
    }
    class FileTxnSnapLog{
        private TxnLog txnLog;      // 记录事务的操作日志的
        private Snapshot sh;        // 针对 DT 拍摄快照的
    }
```

4. 本次课程内容预告

今天的主要内容有：

```
第一部分：基础设施
    1、ZooKeeper版本选择
    2、Zookeeper源码环境准备
    3、ZooKeeper序列化机制
    4、ZooKeeper持久化机制
    5、ZooKeeper网络通信机制
    6、ZooKeeper的Watcher工作机制
第二部分：核心工作流程
    7、ZooKeeper的集群启动脚本分析
    8、ZooKeeper的QuorumPeerMain启动分析
    9、Zookeeper冷启动数据恢复
    10、ZooKeeper选举FastLeaderElection源码剖析
```

5. ZooKeeper 源码剖析

5.1. ZooKeeper版本选择

你为什么需要看源码呢？两大看源码的需求支撑：

```
1、企业需求：你的项目遇到了困难，看源码解决
2、兴趣爱好 + 为了面试
```

zookeeper的大版本：

```
1、zookeeper-3.4.x  企业最常用，大数据技术组件最常用，基本维持在 3.4.5  3.4.6  3.4.7 这几个版本
2、zookeeper-3.5.x
3、zookeeper-3.6.x
```

最总结论：zookeeper-3.4.14.tar.gz，安装包就是源码包

ZooKeeper-3.5 以上，源码 和 安装包就分开了。

整体的原则：不新不旧的稳定版本 + 考虑企业使用版本

5.2. ZooKeeper 源码环境准备

不需要过多的准备，准备一个 IDE，从官网下载源码包，然后直接用 IDE 打开即可！

```
1、准备一个IDE：IDEA
2、从官网下载源码包，IDEA去导入这个源码项目即可
3、稍微等待一下，maven去下载一些依赖jar
```

下载源码的方式：

```
1、从官网下载 zookeeper-3.4.14.tar.gz 安装包，该安装包直接包含源码
2、从 github 去拉取源码项目
```

5.3. ZooKeeper 序列化机制

到底在那些地方需要使用序列化技术呢?

```
1、当在网络中需要进行消息，数据，等的传输，那么这些数据就需要进行序列化和反序列化
2、当数据需要被持久化到磁盘的时候。
```

ZooKeeper（分布式协调服务组件 + 存储系统）

任何一个分布式系统的底层，都必然会有网络通信，这就必然要提供一个分布式通信框架和序列化机制。所以我们在看 ZooKeeper 源码之前，先搞定 ZooKeeper 网络通信和序列化。

```
三种：
1、java提供的序列化机制
2、hadoop的序列化技术
3、zookeeper的序列化
```

将来在阅读 Spark 源码的时候，再给大家讲解spark的序列化机制!

1、Java序列化机制（序列化过程中：类型信息，+ 对象实例的属性值）

特点就是比较笨重：（除了实例的属性信息以外，还会序列化这个实例的类型信息）

Spark 默认使用的序列化机制就是 Java 原生序列化机制， 也提供其他的序列化方式

```
class Student implements Serializable
```

使用 ObjectInputStream 和 ObjectOutputStream 来进行具体的序列化和反序列化。

2、Hadoop中的序列化：

```
class Student implements Writable{

    // 反序列化
    void readFields(DataIn input);

    // 序列化
    void write(DataOut output);
}
```

3、ZooKeeper 中的序列化机制：

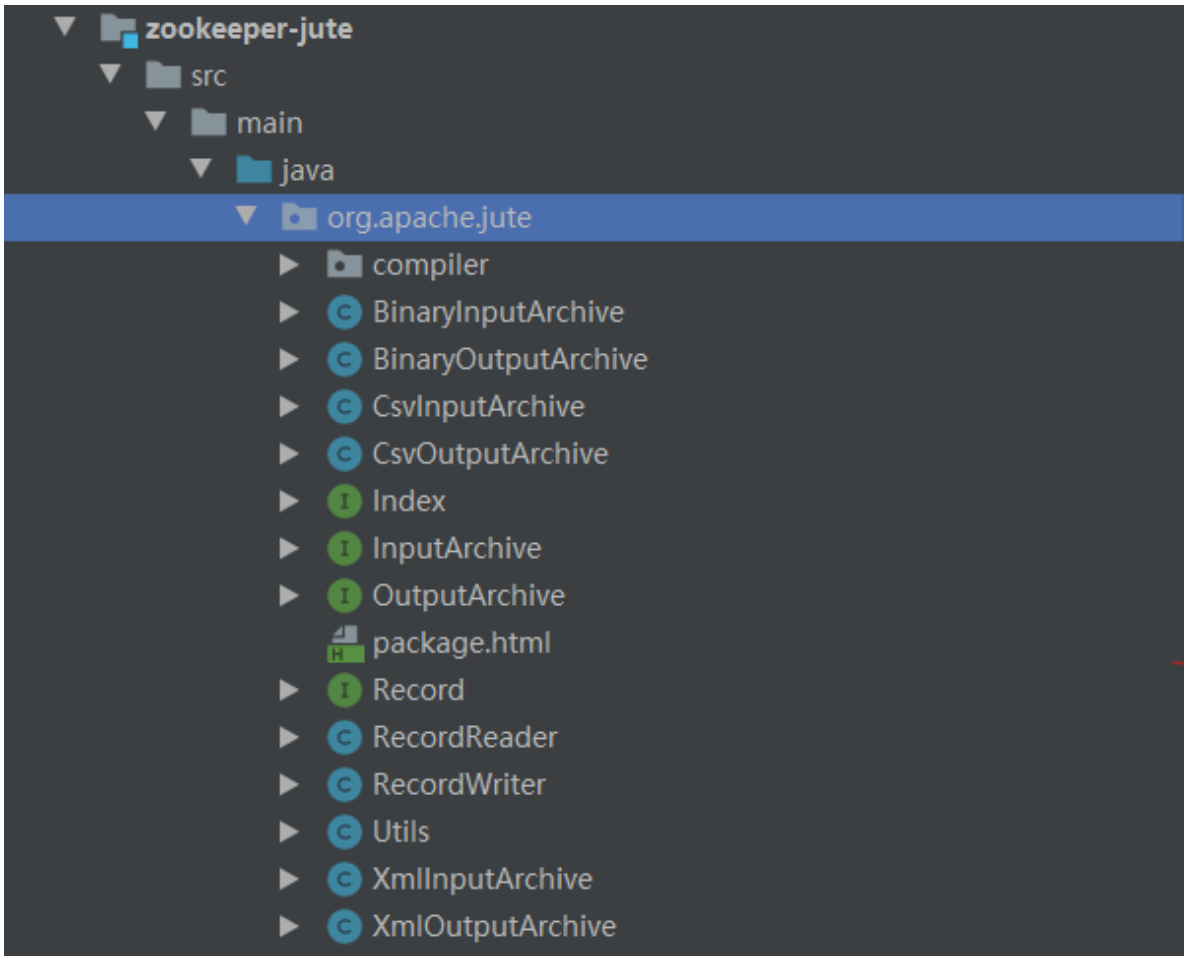
```
class Student implements Record{

    // 反序列化
    void deserialize(InputArchive archive, String tag){
        archive.readBytes();
        archive.readInt();
    }

    // 序列化
    void serialize(OutputArchive archive, String tag)
}
```

```
一个自定义类的实现，都是有多个普通数据类型的属性组成的!
class Student{
    private int id;
    private String name;
    .....
}
```

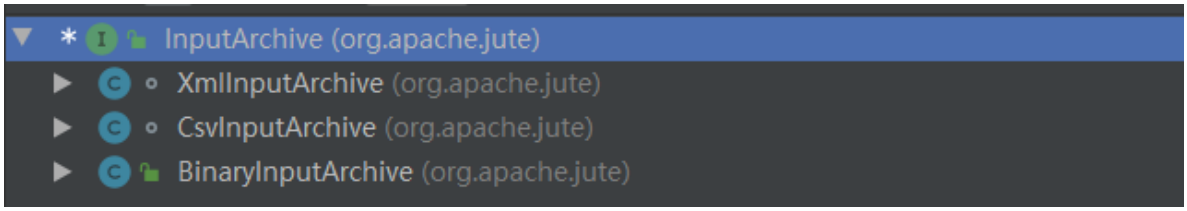
序列化的 API 主要在 zookeeper-jute 子项目中。



重点API：

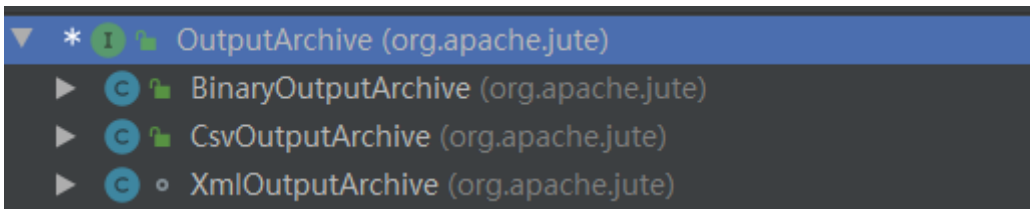
org.apache.jute.InputArchive：反序列化需要实现的接口，其中各种 read 开头的方法，都是反序列化方法

实现类有:



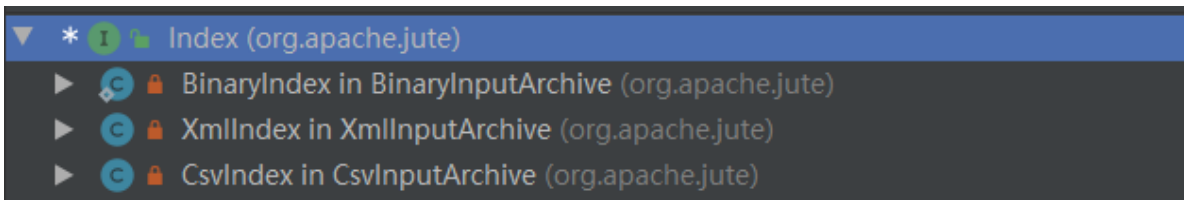
org.apache.jute.OutputArchive：所有进行序列化操作的都是实现这个接口，其中各种 write 开头的方法都是序列化方法。

实现类有：



org.apache.jute.Index：用于迭代数据进行反序列化的迭代器

实现类有：



org.apache.jute.Record：在 ZooKeeper 要进行网络通信的对象，都需要实现这个接口。里面有序列化和反序列化两个重要的方法

5.4. ZooKeeper 持久化机制

```
只要底层涉及到关于数据的存储/读写操作！一般来说，都会有一个持久化机制！
HDFS,Kafka,HBase,ZooKeeper,....
```

ZooKeeper 的数据模型主要涉及两类知识：**数据模型** 和 **持久化机制**，其实就是两套 API

```
1、数据模型：DataTree + DataNode
2、持久化机制：FileTxnSnalLog = TxnLog + SnapLog
3、zk数据库：ZKDataBase = DataTree + FileTxnSnalLog
```

在 ZKDataBase 对象的内部有两个成员变量：DataTree + FileTxnSnalLog

ZooKeeper 本身是一个对等架构（内部选举，从所有 learner 中选举一个 leader， 剩下的成为 follower）

```
1、每个节点上都保存了整个系统的所有数据(leader存储了数据，所有的follower节点都是leader的副本节点)
2、每个节点上的都把数据放在磁盘一份，放在内存一份
```

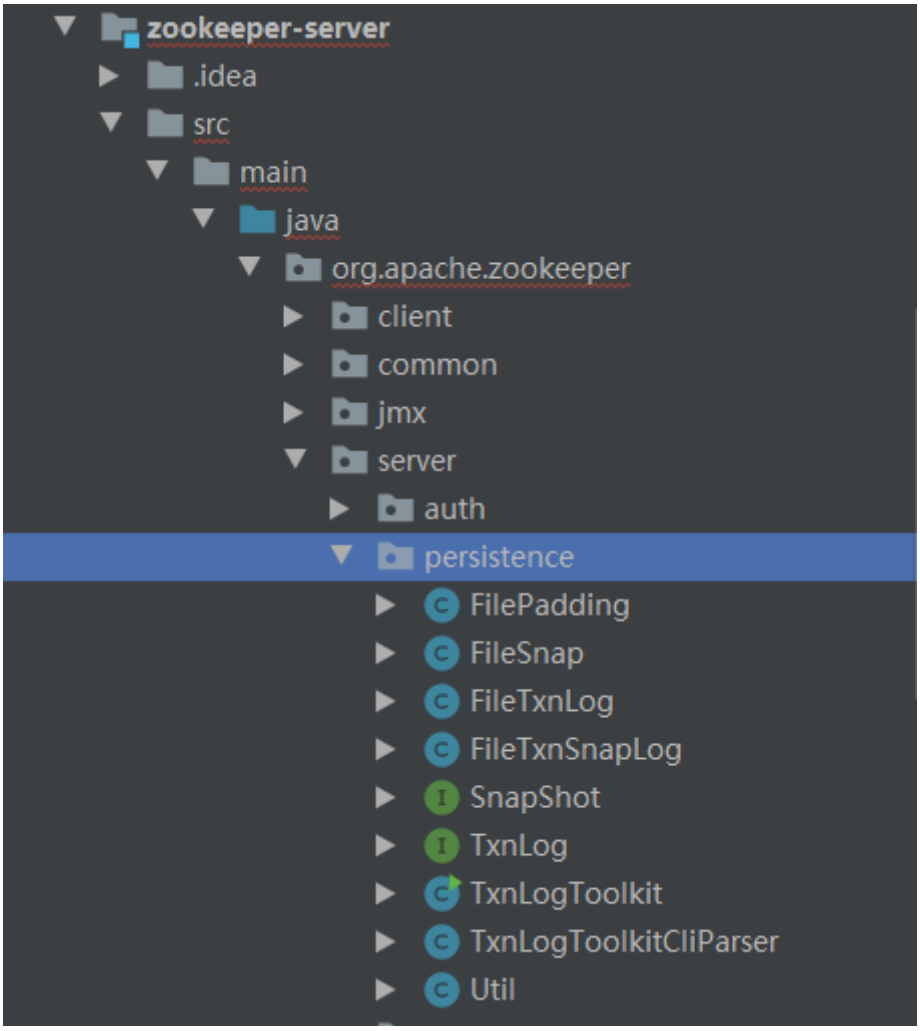
ZooKeeper的数据模型，抽象出了重要的三个API用来完成数据的管理：

```
1、DataNode        znode 系统中的一个节点的抽象
2、DataTree        znode系统的完整抽象
3、ZKDataBase      负责管理 DataTree ，执行 DataTree 的相关 快照和恢复的操作
```

关于 ZooKeeper 中的数据在内存中的组织，其实就是一棵树：

```
1、这棵树就叫做：DataTree  （抽象了一棵树）
2、这棵树上的节点：DataNode （抽象一个节点）
3、关于管理这个 DataTree 的组件就是 ZKDataBase （内存数据库：针对 DataTree 能做各种操作）
```

ZooKeeper 的持久化的一些操作接口，都在：org.apache.zookeeper.server.persistence 包中。



主要的类的介绍：

```
第一组：主要是用来操作日志的（如果客户端往zk中写入一条数据，则记录一条日志）
    TxnLog，接口，读取事务性日志的接口。
    FileTxnLog，实现TxnLog接口，添加了访问该事务性日志的API。

第二组：拍摄快照（当内存数据持久化到磁盘）
    Snapshot，接口类型，持久层快照接口。
    FileSnap，实现Snapshot接口，负责存储、序列化、反序列化、访问快照。

第三组：两个成员变量：TxnLog和SnapShot
    FileTxnSnapLog，封装了TxnLog和SnapShot。

第四组：工具类
    Util，工具类，提供持久化所需的API。
```

5.5. ZooKeeper 网络通信机制

Java IO 有几个种类：(百度搜索；五种IO模型)

```
1、BIO JDK-1.1(编码简单，效率低) 阻塞模型
    选举过程中，多个节点之间的相互通信使用的网络通信模型
2、NIO JDK-1.4(效率有提升，编码复杂) 基于reactor实现的异步非阻塞网络通信模型
    通常的IO的选择：
    1、原生NIO
    2、基于NIO实现的网络通信框架：netty
3、AIO JDK-1.7(效率最高，编码复杂度一般) 真正的异步非阻塞通信模型
```

NIO 的三大API：

```
1、Buffer
2、Channel
3、Selector
```

ZooKeeper 中的通信有两种方式：

```
1、NIO，默认使用NIO
2、Netty，未来版本
```

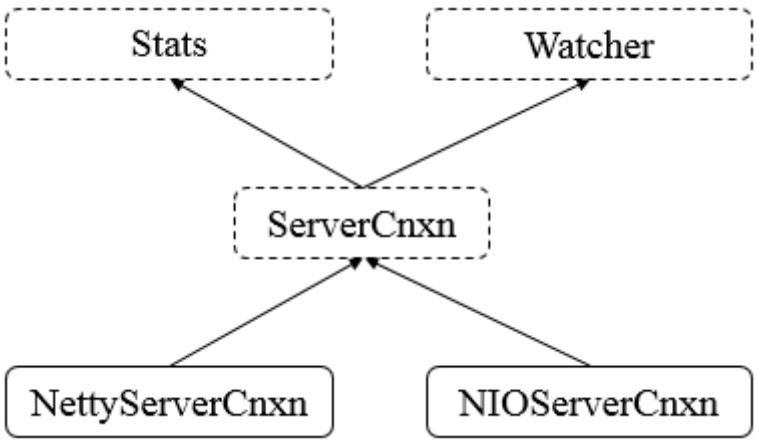
两个最重要的API： C/S 架构模式（）

```
1、ServerCnxn        服务端的通信组件
    zookeeper集群启动的时候，会初始化 ServerCnxn
2、ClientCnxn        客户端的通信组件
    当编写代码：new ZooKeeper(1,2,3) 底层会初始化一个通信客户端对象
```

关于客户端和服务端的一个定义：谁发请求，谁就是客户端，谁接收和处理请求，谁就是服务端

```
1、真正的client给zookeeper发请求
2、zookeeper中的leader给follower发命令
3、zookeeper中的followe给leader发请求
```

ServerCnxn：org.apache.zookeeper.server.ServerCnxn



详细说明：

```
Stats，表示ServerCnxn上的统计数据。
Watcher，表示事件处理，监听器
ServerCnxn，表示服务器连接，表示一个从客户端到服务器的连接。
ClientCnxn，存在于客户端用来执行通信的组件
NettyServerCnxn，基于Netty的连接的具体实现。
NIOServerCnxn，基于NIO的连接的具体实现。
ServerCnxnFactory，服务端通信组件的工厂实现 也有两种：
            NIOServerCnxnFactory + NettyServerCnxnFactory
```

5.6. Zookeeper的Watcher工作机制

客户端的 Watcher 注册：

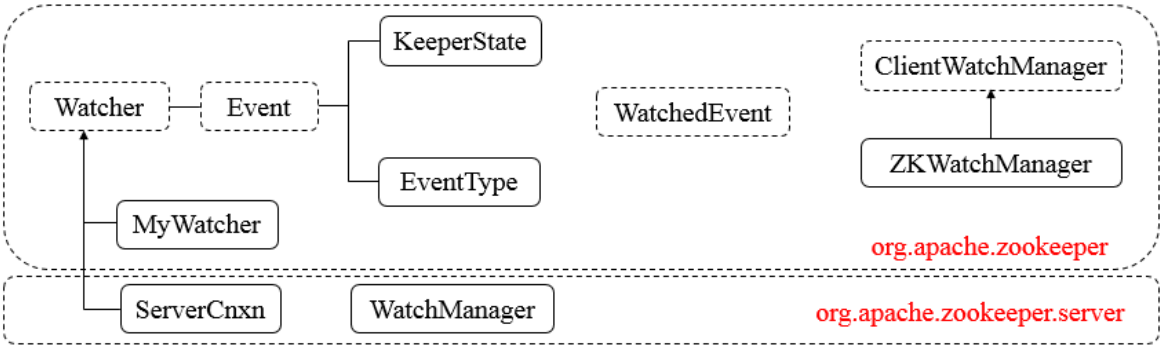
```
1、org.apache.zookeeper.ZooKeeper：客户端基础类、存储了ClientCnxn和ZkWatcherManager

2、ZKWatchManager：ZooKeeper的内部类，实现了ClientWatchManager接口，主要用来存储各种类型的Watcher，主要有三种：dataWatches、existWatches、childWatches以及一个默认的defaultWatcher

3、org.apache.zookeeper.ClientCnxn：与服务端的交互类，主要包含以下对象：LinkedListoutgoingQueue、pendingQueue，SendThread 和 EventThread，其中outgoingQueue未待发送给服务端的Packet列表，SendThread线程负责和服务端进行请求交互，而EventThread线程则负责客户端Watcher事件的回调执行

4、WatchRegistration：Zookeeper的内容类，包装了Watcher和clientPath，并且负责Watcher的注册

5、Packet：ClientCnxn的内部类，与Zookeeper服务端通信的交互类，ClientCnxn 和 ServerCnxn 进行通信的时候，发送的都是 Packet
```



两条主线

```
1、实现主线：Watcher + WatchedEvent
2、管理主线：WatchManager（负责响应：watcher.process(watchedEvent)） + ZKWatchManager（负责注册等相关管理）
```

```
interface Watcher{
    interface Event{
        enum KeeperState        链接状态
        enum EventType          事件类型
    }
```

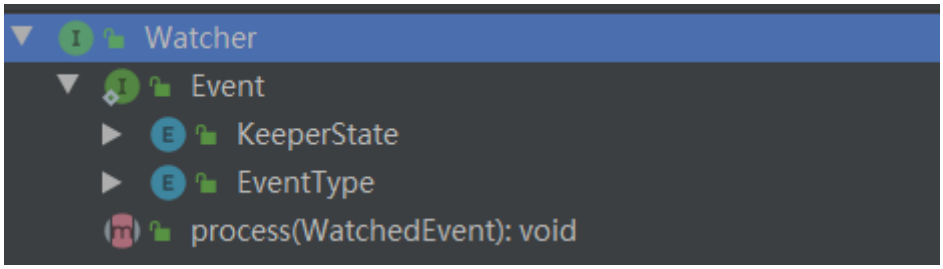
```
    // 这就是回调方法（触发的事件：KeeperState, znodePath, EventType）
    void process(WatchedEvent event)
}

// 表示触发了一次监听事件的一个响应对象：链接状态 + znode节点路径 + 操作事件
class WatchedEvent{
    KeeperState state     会话连接的状态信息
    String path          znode节点的绝对路劲
    EventType type       事件的类型
}
```

组件说明：

```
Watcher，接口类型，其定义了process方法，需子类实现。
Event，接口类型，Watcher的内部类，无任何方法。
KeeperState，枚举类型，Event的内部类，表示Zookeeper所处的状态。
EventType，枚举类型，Event的内部类，表示Zookeeper中发生的事件类型。
WatchedEvent，表示对ZooKeeper上发生变化后的反馈，包含了KeeperState和EventType。
ClientWatchManager，接口类型，表示客户端的Watcher管理者，其定义了materialized方法，需子类实现。
ZKWatchManager，Zookeeper的内部类，继承ClientWatchManager。
MyWatcher，ZooKeeperMain的内部类，继承Watcher。
ServerCnxn，接口类型，继承Watcher，表示客户端与服务端的一个连接。
WatchManager，管理Watcher。
```

Watcher类组成：



WatchedEvent构成：

```
/**
 *  A WatchedEvent represents a change on the ZooKeeper that a Watcher
 *  is able to respond to.  The WatchedEvent includes exactly what happened,
 *  the current state of the ZooKeeper, and the path of the znode that
 *  was involved in the event.
 */
@InterfaceAudience.Public
public class WatchedEvent {

    // 链接信息
    final private KeeperState keeperState;
    // 事件类型
    final private EventType eventType;
    // 事件发生的znode节点
    private String path;
```

Watcher 主要工作流程：

1. 用户调用 Zookeeper 的 getData 方法，并将自定义的 Watcher 以参数形式传入，该方法的作用主要是封装请求，然后调用 ClientCnxn 的 submitRequest 方法提交请求

2. ClientCnxn 在调用 submitRequest 提交请求时，会将 WatchRegistration(封装了我们传入的 Watcher 和clientPath)以参数的形式传入，submitRequest 方法主要作用是将信息封装成 Packet(ClientCnxn的内部类)，并将封装好的 Packet 加入到 ClientCnxn 的待发送列表中 (LinkedList outgoingQueue)
3. SendThread 线程不断地从 outgoingQueue 取出未发送的 Packet 发送给客户端并且将该 Packet 加入pendingQueue (等待服务器响应的Packet列表)中，并通过自身的 readResponse 方法接收服务端的响应
4. SendThread 接收到客户端的响应以后，会调用 ClientCnxn 的 finishPacket 方法进行 Watcher 方法的注册
5. 在 finishPacket 方法中，会取出 Packet 中的 WatchRegistration 对象，并调用其 register 方法，从ZKWatchManager 取出对应的 dataWatches、existWatches 或者 childWatches 其中的一个 Watcher 集合，然后将自己的 Watcher 添加到该 Watcher 集合中。

5.7. HDFS源码

```
1、HDFS 架构的演进
2、集群启动
    namenode启动
    datanode启动
3、核心工作机制
    上传
    下载
```

ZK 总结成两个方面：

```
1、集群启动好之前
    集群启动脚本分析 zkServer.sh start
    集群启动的启动类的代码执行分析  QuorumPeerMain.main()
    冷启动数据恢复，从磁盘恢复数据到内存  zkDatabase.loadDatabase()
    选举       startLeaderElection() + QuorumPeer.lookForLeader()
    同步       follower.followLeader() + observer.observerLeader()      下一次课
2、集群启动好之后
    读写处理
    ZooKeeper zk = new ZooKeeper("bigdata02:2181", 4000, watcher)
    zk.create(节点路径，节点数据，节点的ACL，节点的类型)
```

5.8. ZooKeeper的集群启动脚本分析

第一个问题；到底哪些源码流程我们需要关注呢？

```
1、集群的启动
2、崩溃恢复（leader选举）+ 原子广播（状态同步）
3、读写请求
```

第二个问题：到底从哪个地方入手看源码？入口

启动 ZooKeeper 的时候：

```
zkServer.sh start
```

底层会转到调用：QuorumPeerMain.main()

具体实现：见文档：ZooKeeper 启动脚本分析

5.9. ZooKeeper的QuorumPeerMain启动

大致流程：

```
# 入口方法
QuorumPeerMain.main();

    # 核心实现，分三步走
    QuorumPeerMain.initializeAndRun(args);

        # 第一步：解析配置
        config = new QuorumPeerConfig();
        config.parse(args[0]);
            Properties cfg = new Properties();
            cfg.load(in);
            parseProperties(cfg);

        # 第二步：启动一个线程（定时任务）来执行关于old snapshot的clean
        new DatadirCleanupManager(...).start()
            timer = new Timer("PurgeTask", true);
            TimerTask task = new PurgeTask(dataLogDir, snapDir,
snapRetainCount);
            timer.scheduleAtFixedRate(task, 0,
TimeUnit.HOURS.toMillis(purgeInterval));
                PurgeTxnLog.purge(new File(logsDir), new File(snapsDir), ...);

        # 第三步：启动（有两种模式：standalone，集群模式）重点关注集群启动，分两步走
        runFromConfig(config);

            # 第一件事：服务端的通信组件 的初始化，但是并未启动
            factory = ServerCnxnFactory.createFactory();
            cnxnFactory.configure(....)

            # 第二件事：抽象一个zookeeper节点，然后把解析出来的各种参数给配置上，然后启动
            quorumPeer = getQuorumPeer(); + quorumPeer.setXXX() +
quorumPeer.start();

                # 第一件事：把磁盘数据恢复到内存
                loadDataBase();
                    zkDb.loadDataBase();
                        # 冷启动的时候，从磁盘恢复数据到内存
                        snapLog.restore(...., ....)
                            # 从快照恢复
                            snapLog.deserialize(dt, sessions);
                            # 从操作日志恢复
                            fastForwardFromEdits(dt, sessions, listener);
                                # 恢复执行一条事务
                                rocessTransaction(hdr, dt, sessions,
itr.getTxn());

                # 第二件事：服务端的通信组件的真正启动
```

```
                cnxnFactory.start();

                # 第三件事：准备选举的一些必要操作(初始化一些队列和一些线程)
                startLeaderElection();

                # 第四件事：调用 start() 跳转到 run() 方法。因为 QuorumPeer被封装成
Thread 了
                super.start();
                    # 执行选举
                    QuorumPeer.run()
                        # 选举入口
                        FastLeaderElection.lookForLeader()
```

zoo.cfg 中的内容：

```
# The number of milliseconds of each tick
tickTime=2000
# The number of ticks that the initial
# synchronization phase can take
initLimit=10
# The number of ticks that can pass between
# sending a request and getting an acknowledgement
syncLimit=5
# the directory where the snapshot is stored.
# do not use /tmp for storage, /tmp here is just
# example sakes.
dataDir=/home/bigdata/data/zkdata
dataLogDir=/home/bigdata/data/zklog/
# the port at which the clients will connect
clientPort=2181
# the maximum number of client connections.
# increase this if you need to handle more clients
#maxClientCnxns=60
# electionAlg=3 = FastLeaderElectioin
electionAlg=3
maxClientCnxns=60
peerType=observer/participant
# Be sure to read the maintenance section of the
# administrator guide before turning on autopurge.
#
# http://zookeeper.apache.org/doc/current/zookeeperAdmin.html#sc_maintenance
#
# The number of snapshots to retain in dataDir
autopurge.snapRetainCount=3
# Purge task interval in hours
# Set to "0" to disable auto purge feature
autopurge.purgeInterval=1

server.2=bigdata02:2888
server.3=bigdata03:2888:3888
server.4=bigdata04:2888:3888
server.5=bigdata05:2888:3888:observer
```

5.10. ZooKeeper的冷启动数据恢复

入口方法：QuorumPeer.loadDataBase();

大致流程：

```
# 入口方法
QuorumPeer.loadDataBase();
    zkDb.loadDataBase();
        # 冷启动的时候，从磁盘恢复数据到内存
        snapLog.restore(dataTree,...)

            # 第一件事：从快照恢复
            snapLog.deserialize(dt, sessions);
                deserialize(dt, sessions, ia);
                    SerializeUtils.deserializeSnapshot(dt, ia, sessions);
                        dt.deserialize(ia, "tree");

            # 第二件事：从操作日志恢复
            fastForwardFromEdits(dt, sessions, listener);
                 # 恢复执行一条事务
                processTransaction(hdr, dt, sessions, itr.getTxn());
                    # 恢复执行一条事务
                    dt.processTxn(hdr, txn);
                        # 创建一个znode
                        createNode 或者 deleteNode
```

详细内容见源码注释。

5.11. ZooKeeper选举FastLeaderElection源码剖析

背景知识：

```
1、所有的节点（有选举权和被选举权），一上线，就是 LOOKING 状态，当选举结束了之后，有选举权中的角色会变量另外两种：Leader, Follower。

2、需要发送选票，选票是 Vote 对象，广播到所有节点。事实上，关于选票和投票的类型有四种：
    Vote选票    Notificatioin接收到的投票    Message放入投票队列的时候会变成   ToSend待发送的选票
    还有一个特殊的中间对象：ByteBuffer   NIO 的一个API

3、当每个 zookeeper 服务器启动好了之后，第一件事就是发起投票，如果是第一次发起投票都是去找 leader，如果发现有其他 zookeeper 返回给我 leader 的信息，那么选举结束。

4、在进行选票发送的时候，每个 zookeeper 都会为其他的 zookeeper 服务节点生成一个对应的 SendWorker 和一个 ArrayBlockingQueue， ArrayBlockingQueue 存放待发送的选票，SendWorker 从队列中，获取选票执行发送。还有一个线程叫做： ReceiveWorker 真正完整从其他节点接收发送过来的投票。
```

入口：startLeaderElection(); 但是一定要记住：该方法只是为选举做准备！

大致过程如下：

```
# 入口方法
startLeaderElection();
```

```
    # 第一步：初始化选票（myid, zxid, epoch）
    currentVote = new Vote(myid, getLastLoggedZxid(), getCurrentEpoch());

    # 第二件事：初始化选举算法
    this.electionAlg = createElectionAlgorithm(electionType);

        # 第一件事：初始化了 QuorumCnxManager
        qcm = createCnxnManager();
            # 调用了 QuorumCnxManager 的构造函数 QuorumCnxManager()
            new QuorumCnxManager(.....)
                # 初始化一个 ConcurrentHashMap<Long, SendWorker> senderWorkerMap
                # 初始化一个 ArrayBlockingQueue<Message> recvQueue
                # 初始化一个 ConcurrentHashMap<Long, 发送队列>() queueSendMap
                # 初始化一个 ConcurrentHashMap<Long, ByteBuffer> lastMessageSent
                # 初始化 Listener, 这个 listener 就是  qcm 的成员变量
                listener = new Listener();

        # 第二件事：启动 QuorumCnxManager.Listener
        Listener.start()
            # 内部启动一个选举的服务端，等待其他zookeeper发送链接请求过来，建立连接，发送选
票
            # 等待有其他的zookeeper节点发送发起选举请求的链接过来
            Listener.run()
                # 启动 BIO 服务端
                ss = new ServerSocket();
                # 在初始化的时候，代码会卡在这儿
                while(!shutdown){Socket client = ss.accept();}
                    # 处理链接请求
                    receiveConnection(client);
                        # 如果对方的 myid比我小，关闭连接，只能 myid 大的向 myid 小的发
起链接
                        if(sid < this.mySid) {
                            # 重新建立连接
                            connectOne(sid);
                                # 初始化 BIO 客户端，然后初始链接
                                Socket sock = new Socket();
                                initiateConnection(sock, sid);
                                    startConnection(sock, sid);
                                        # 初始化 选票发送线程 和 投票接收线程
                                        SendWorker sw = new SendWorker(sock,
sid);
                                        RecvWorker rw = new RecvWorker(....);
                                        sw.start();
                                        rw.start();

                        }else{
                            # 处理一个链接
                            handleConnection(sock, din);
                                SendWorker sw = new SendWorker(sock, sid);
                                RecvWorker rw = new RecvWorker(sock, din, sid,
sw);
                                sw.start();
                                rw.start();
                        }

        # 第三件事：初始化选举算法
        new FastLeaderElection(this, qcm);
            starter(self, manager);
```

```
                # 第一件事：  统一发送队列
                sendqueue = new LinkedBlockingQueue<ToSend>();
                # 第二件事：  统一接收队列
                recvqueue = new LinkedBlockingQueue<Notification>();
                # 第三件事：  统一发送和接收线程！
                this.messenger = new Messenger(manager);
                    this.ws = new WorkerSender(manager);
                    ws.start()
                    this.wr = new WorkerReceiver(manager);
                    wr.start()
```

终于启动了选举：入口方法：FastLeaderElection.lookForLeader(); 这才是真正开始选举

```
# 入口，当调用 QuorumPeer 的 start() 方法的时候，它内部的 super.start() 会转到 run 方法
QuorumPeer.run()
    # 真正负责选举的是 FastLeaderElection
    FastLeaderElection.lookForLeader()
        # 初始化存储 合法选票的 容器
        HashMap<Long, Vote> recvset = new HashMap<Long, Vote>();
        # 自增逻辑时钟
        logicalclock.incrementAndGet();
        # 准备选票
        updateProposal(getInitId(), getInitLastLoggedZxid(), getPeerEpoch());
        # 广播选票
        sendNotifications();
            # 将选票存入发送队列
            sendqueue.offer(notmsg);
                # 处理选票的发送
                WorkerSender.run() + process(m);
                    manager.toSend(m.sid, requestBuffer);
                        # 将选票放置对应server的发送队列
                        addToSendQueue(bqExisting, b);
                             queue.add(buffer);  + SendWorker
                        # 发起和该服务器的链接请求，
                        connectOne(sid);
                              Socket sock = new Socket();
                              sock.connect(view.get(sid).electionAddr, cnxTO);
                              # 当我初始化一个 BIO 客户端，然后发起链接
                              # 我自己也会初始化链接(初始化SendWorker和RecvWorker )
                              # 对方也会接收到请求，然后初始化链接(初始化SendWorker和
RecvWorker)
                              initiateConnection(sock, sid);
                                    # 初始化 发送选票的 线程
                                    SendWorker sw = new SendWorker(sock, sid);
                                    # 初始化 接收投票的 线程
                                    RecvWorker rw = new RecvWorker(sock, din,
sid, sw);
                                        SendWorker.run()
                                            # 发送选票
                                            send(b)
                                        RecvWorker.run()
                                            # 等接收投票
                                            din.readFully(msgArray, 0, length);
                                            addToRecvQueue(message)
                                                # 将投票结果放置在接收队列recvQueue
中
                                                recvQueue.add(msg);
```

换一种中文解释：Leader选举的基本流程如下

1. **自增选举轮次。**Zookeeper 规定所有有效的投票都必须在同一轮次中，在开始新一轮投票时，会首先对 logicalclock 进行自增操作。
2. **初始化选票。**在开始进行新一轮投票之前，每个服务器都会初始化自身的选票，并且在初始化阶段，每台服务器都会将自己推举为 Leader。
3. **发送初始化选票。**完成选票的初始化后，服务器就会发起第一次投票。Zookeeper 会将刚刚初始化好的选票放入 sendqueue 中，由发送器 WorkerSender 负责发送出去。
4. **接收外部投票。**每台服务器会不断地从 recvqueue 队列中获取外部选票。如果服务器发现无法获取到任何外部投票，那么就会立即确认自己是否和集群中其他服务器保持着有效的连接，如果没有连接，则马上建立连接，如果已经建立了连接，则再次发送自己当前的内部投票。
5. **判断选举轮次。**在发送完初始化选票之后，接着开始处理外部投票。在处理外部投票时，会根据选举轮次来进行不同的处理。

- **外部投票的选举轮次大于内部投票。**若服务器自身的选举轮次落后于该外部投票对应服务器的选举轮次，那么就会立即更新自己的选举轮次(logicalclock)，并且清空所有已经收到的投票，然后使用初始化的投票来进行PK以确定是否变更内部投票。最终再将内部投票发送出去。
- **外部投票的选举轮次小于内部投票。**若服务器接收的外选票的选举轮次落后于自身的选举轮次，那么Zookeeper就会直接忽略该外部投票，不做任何处理，并返回步骤4。
- **外部投票的选举轮次等于内部投票。**此时可以开始进行选票PK。

6. **选票PK。**在进行选票PK时，符合任意一个条件就需要变更投票。

- 若外部投票中推举的Leader服务器的选举轮次大于内部投票，那么需要变更投票。
- 若选举轮次一致，那么就对比两者的ZXID，若外部投票的ZXID大，那么需要变更投票。
- 若两者的ZXID一致，那么就对比两者的SID，若外部投票的SID大，那么就需要变更投票。

7. **变更投票。**经过PK后，若确定了外部投票优于内部投票，那么就变更投票，即使用外部投票的选票信息来覆盖内部投票，变更完成后，再次将这个变更后的内部投票发送出去。
8. **选票归档。**无论是否变更了投票，都会将刚刚收到的那份外部投票放入选票集合 recvset 中进行归档。recvset 用于记录当前服务器在本轮次的 Leader 选举中收到的所有外部投票（按照服务队的SID区别，如{(1, vote1), (2, vote2)...}）。
9. **统计投票。**完成选票归档后，就可以开始统计投票，统计投票是为了统计集群中是否已经有过半的服务器认可了当前的内部投票，如果确定已经有过半服务器认可了该投票，则终止投票。否则返回步骤4。
10. **更新服务器状态。**若已经确定可以终止投票，那么就开始更新服务器状态，服务器首选判断当前被过半服务器认可的投票所对应的Leader服务器是否是自己，若是自己，则将自己的服务器状态更新为 LEADING，若不是，则根据具体情况来确定自己是 FOLLOWING 或是 OBSERVING。

以上 10 个步骤就是 FastLeaderElection 的核心，其中步骤 4-9 会经过几轮循环，直到有 Leader 选举产生

6. 总结

今天讲述的内容，主要是：ZooKeeper 的基础设施组件 和 集群启动源码剖析。主要涉及到启动流程分析，冷启动数据恢复，和选举三大核心操作。